PATH DELAY TEST THROUGH MEMORY ARRAYS

A Thesis

by

PUNJ POKHAREL

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Duncan M. H. Walker
Committee Members,	Rabi N. Mahapatra
	Paul Gratz
Head of Department,	Duncan M. H. Walker

August 2013

Major Subject: Computer Engineering

ABSTRACT

Memory arrays cannot be as easily tested as other storage elements in a chip. Most of the flip-flops (FFs) in a chip can be replaced by scan cells in scan-based design. However, the bits in memory arrays cannot be replaced by scan cells, due to the area cost and the timing-critical nature of many of the paths into and out of memories. Thus, bits in a memory array can be considered non-scan storage elements.

Test methods such as memory built-in self-test (MBIST), functional test, and macro test are used to test memory arrays. However, these tests aren't sufficient to test the paths through the memory arrays. During structural (scan) test generation, memory arrays are treated as "black boxes" or memory arrays are bypassed to a known value. Black boxes decrease coverage loss while bypassing increases chip area and delay.

Path delay test through memory arrays is proposed using pseudo functional test (PFT) with K Longest Paths Per Gate (KLPG). In this technique, any longest path that is captured into a non-scan cell (including a memory cell) is propagated to a scan cell. The propagation of the captured value from non-scan cell to scan cell occurs during low-speed clock cycles. In this work, we assume that only one extra *coda* cycle is sufficient to propagate the captured value to a scan cell. This is true if the output of the memory feeds combinational logic that in turn feeds scan cells. When we want to launch a transition from a memory output, different values are written into different address locations and the address is toggled between the locations. The ATPG writes the different values into the memory cells during the preamble cycles. In the case of

launching a transition out of a non-scan cell, the cell must be written with an initial value during the preamble cycles, and the next value set on the non-scan cell input. Thus, it is possible to capture and launch transitions into and from memory and non-scan cells and thus test the path delay of the longest paths into and out of memory and non-scan cells.

ACKNOWLEDGEMENTS

I would like to thank my M.S. advisory committee chair Dr. Duncan M. "Hank" Walker, and my committee members, Dr. Rabi N. Mahapatra and Dr. Paul Gratz, for their guidance and support throughout the course of this research. I would also like to thank Tengteng Zhang for his continuous support and company in research and in the lab.

I am grateful to all the staff, faculty and friends who have helped me in various ways in the course of time. I would like to thank Michael Mateja for his guidance during my internship at Advanced Micro Devices in Austin TX.

Finally, thanks to my family for what they have given to me.

NOMENCLATURE

ATPG	Automatic Test Pattern Generation
DFT	Design For Test
LOC	Launch On Capture
PFT	Pseudo Functional Test
KLPG	K Longest Path Per Gate
SCOAP	Sandia Controllability/Observability Analysis Program
PSN	Power Supply Noise
PI	Primary Input
PO	Primary Output
PPI	Pseudo Primary Input
PPO	Pseudo Primary Output

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

1. INTRODUCTION

1.1 Memory tests

Most of the storage elements in a chip are replaced by scan cells in a scan based design. These scan cells have greater area than the normal storage elements. Most of the scan cells are also accompanied by a multiplexer in a muxed-scan design. There will be an increase in delay for any paths passing through such cells. If timing-critical paths pass through any storage elements, then such elements are not replaced by scan cells. Since most of the critical paths of the chip pass through memory arrays, cells of the memory arrays cannot be replaced by scan cells [1, 2]. The chip area and power consumption increase from replacing memory cells with scan cells would also be unacceptable.

Since the memory cells act as non-scan storage elements and do not form part of a scan chain, we cannot easily launch transitions from memory cells and cannot observe any captured values in the memory. Various techniques have been developed to test memory arrays.

1.1.1 Direct memory access

This technique uses direct access to the memory arrays using the signal pins [3]. This is the easiest technique, but it has many drawbacks. It is not possible to use a large number of pins solely for memory test. This technique can practically only be used in a standalone memory and if there are large memories near chip edge.

Scan chains or TAMs (test access methods) can also be used to deliver test patterns to and results from memories inside a chip. These methods can be scaled to multiple memories on a chip, but cannot provide a delay test.

1.1.2 MBIST

Memory BIST (built-in self-test) or MBIST is the most popular technique in which the memory is tested using functional patterns. Memory BIST is an added DFT structure which has its ports connected to the memory ports as shown in Figure 1.

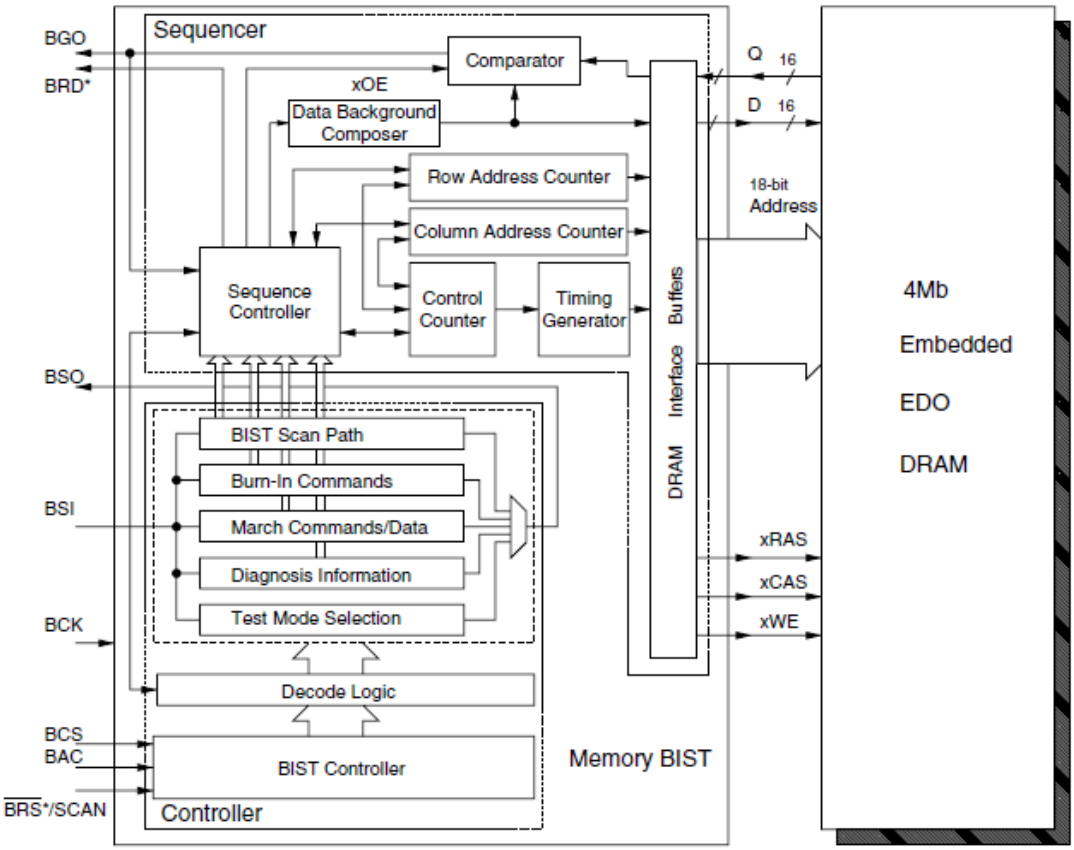

Figure 1 MBIST [4]

MBIST can be used to detect defects such as RAM dynamic faults, stuck-at faults and transition faults. MBIST also supports path delay test if the test is done using an at-speed clock. March patterns, which will be described later in the chapter, are generated by the MBIST and applied to the memory [4]. The response is captured back and is verified against the expected response.

1.1.3 Black box

This technique is used during scan test generation where each memory array is modeled as a "black box." The black box has input and output ports but no internal description is defined. Thus, any captured transitions are lost and no transitions are launched as only unknown (X) values are obtained from the black box. This technique creates "shadow" regions around the memory that cannot be tested, reducing fault coverage.

1.1.4 Memory bypassing

The loss in fault coverage due to black boxes can be compensated by the use of memory bypassing. In this technique the output of the memory is fixed to a certain value or the input into the memory is transferred to the output. This way Xs are prevented from being propagated into the circuit and any memory inputs are captured. However, this technique does not fully test the memory operation, so a separate memory test is required. If the bypass logic makes use of memory circuits, it can provide a partial delay test and produce power supply noise (PSN) similar to functional memory operation. For example, in bypass mode the address decoder could always select a special bypass word in the memory.

1.2 Memory fault model for functional patterns

Memory fault models for functional test are broadly classified into three categories [4, 5, 6]:

- Address Decoder Faults
- Memory Faults
- Dynamic Faults

a) Address Decoder Faults

Address decoder faults (AFs) can be categorized as follows according to their functional behavior: (1) no cell can be accessed by a certain address; (2) multiple cells are accessed simultaneously by a certain address; (3) a certain cell is not accessible by any address; and (4) a certain cell is accessible by multiple addresses. As to the read/write circuitry (including buses, sense amplifiers, and write buffers), the typical faults are equivalent to faults in the memory cell array [4].

b) Memory Faults

There are various memory faults present in memory. A few of the faults are described below [4]:

- Stuck-at fault (SAF) — A cell is stuck-at-1 or 0
- Stuck-open fault (SOF) — A cell is not accessible due to, *e.g.*, a broken word line or a permanently open switch.

- Transition fault (TF) — A cell fails to transit i.e. if there is a rising transition, the cell will have a stable 0 and if there is a falling transition, the cell will have stable 1.
- Data retention fault (DRF) — A cell fails to retain its logic value after a pre-specified period of time.
- Coupling faults
    a. Inversion coupling fault (CFin) — A transition in one cell inverts the content of another.
    b. Idempotent coupling fault (CFid) — A transition in one cell forces a constant value (1 or 0) into another.
    c. State coupling fault (CFst) — A coupled cell or line is forced to a certain value only if the coupling cell or line is in a given state
    d. Read disturb fault (RDF) — The cell value will flip when being read (repeatedly) [7].

c) Dynamic Faults

A static fault is one that has a static behavior; that is, its behavior does not change over time. A dynamic fault, on the other hand, has a dynamic behavior that may change over time [4]. There are three categories of dynamic faults:

- Recovery fault
- Retention fault
- Imbalance fault

a) **A recovery fault** occurs when some part of the memory cannot recover fast enough from a previous state. Popular recovery faults include: (1) sense amplifier recovery fault—the sense amplifier saturates after reading or writing a long string of 0's or 1's; and (2) write recovery fault—a write followed by a read or write at a different location result in reading or writing at the same location due to slow address decoder.

b) **A retention (refresh) fault** occurs when the memory loses its content spontaneously, not caused by the read or write operation. One example is the sleeping sickness of MOS DRAM that is caused by, for example, charge leakage or environment sensitivity, where the DRAM cells lose information in less than the specified hold (refresh) time—typically tens to hundreds of milliseconds. The problem usually affects a row or a column. Another example is the refresh line stuck-at fault, which also can damage the refresh mechanism of the DRAM. For SRAM, there can also be retention faults, caused by a defective pull-up device that induces excessive leakage currents which can change the state of a cell.

c) **Imbalance fault** is the fault where the voltage imbalance between the complementary bit line voltages causes read errors.

1.3 Motivation

Embedded memories occupy a large amount of area in modern chips, and the trend is for increasing amounts of memory. In many designs, the critical timing paths in

the circuit pass through these embedded memories. It is necessary to test these paths in order to accurately determine the clock frequency of the circuit.

Traditional delay testing of integrated circuits uses the transition fault, where a slow-to-rise or slow-to-fall fault exists at a gate input or output. These faults are detected by propagating a transition through the fault site, and capturing the result at a scan cell. The transition fault model is suitable when there is a large delay increase at a location in the chip. However, this fault model may miss localized small delay defects or distributed delay. These more subtle delay defects are increasingly important.

Path delay faults are one of the various delay fault models. In this type of fault model, the delay is accumulated along the path, in contrast to a transition fault model where the delays are accumulated at a node. Thus a large number of paths must be tested for the path delay fault model.

The path delay fault model can be used to detect small delay defects as well as the slowest path in the circuit because a path with low timing slack may fail if it has a small delay defect. This fault model is used in finding the critical path of the circuit and also in speed binning.

The transition fault model and traditional memory test does not have a good correlation with FMAX (the maximum operating frequency of the chip). Path delay test through memory arrays is shown to have a good correlation with FMAX [1, 2]. Thus, path delay test is used in this work on delay testing embedded memory arrays.

The path delay fault will detect small delay defects, as well as defects that can be modeled by transition faults. Since the number of paths in a circuit is potentially

exponential in the number of gates, it is not possible to test all paths. The KLPG algorithm was developed to test the K longest paths through each line in the circuit, limiting the number of paths that must be tested, but providing good coverage of defects.

In both transition and path delay test, complex sequential elements such as memory arrays have been modeled as a "black box", bypassing the memory or via MBIST as already discussed above. However, these techniques either make it impossible to test paths through memory arrays or do not target the longest paths into and out of the memories or have a limited correlation with functional mode, thus precluding the testing of small delay defects in and around the memory. In order to ensure high fault coverage of path and small delay defects in and around the memory, PFT KLPG tests must be performed on the paths into and out of the memory.

1.4 Previous work

Much prior work has been done on memory array test. However, most of this work has focused on functional test or MBIST. Memory arrays can be tested using functional March patterns [8, 9]. MBIST can be used to implement March patterns on chip [1, 10, 11, 12]. Different types of MBIST are available, such as programmable MBIST [5], and multiple BIST controllers for multiple embedded memories [12]. An Embedded Micro-Tester can be used to test the components of an SoC [13] at speed, similar to MBIST. However, these techniques do not guarantee coverage of small delay defects.

Classical March pattern algorithms are described in [4, 5, 7]. Operation in March patterns are represented by three symbols; ⇑ denotes increasing address sequence, ⇓ denotes decreasing address sequence and ↕ denotes increasing or decreasing address sequence. A ‘w’ denotes write, a ‘r’ denotes read, and a 0/1 denotes the value written or expected.

The **Zero-one algorithm** writes 0 in all cells and reads them and writes 1 in all cells and reads them again. So it is described by {⇑w0; ⇑r0; ⇑w1; ⇑r1}. This algorithm is also known as the MSCAN algorithm. The algorithm can also be given as follows:

Procedure ZERO-ONE
{

1: write 0 in all cells;

2: read all cells;

3: write 1 in all cells;

4: read all cells;

}

Another March algorithm is the **Checkerboard algorithm**, which writes 0s and 1s in alternate bits or words of the memory array so that the overall memory array looks like a checkerboard. The checkerboard pattern is mainly used for activating failures resulting from, for example, leakage, shorts between cells, and data retention, though it also detects stuck-at faults and half of the transition faults. The checkerboard algorithm is described below:

Procedure Checkerboard

```
{
        while (i is odd && j is even)
        { write 0 in cell[i]; write 1 in cell[j];
        pause; read all cells;
        complement all cells;
        pause; read all cells; }
}
```

The **Galloping pattern (GALPAT) algorithm** reads the base cell alternately with every other cell in its set – the entire cell array [4, 5, 7]. It is a strong test for most faults — all Address Decoder Faults, Transition Faults, Coupling Faults and Stuck-At Faults are detected and located. Since the time is quadratic in memory size, instead of all cells in the array, the set may also be a column, a row, or a diagonal. The algorithm is described below:

```
Procedure GALPAT
{
        1: write 0 in all cells;
        2: for i = 0 to n−1
        {
        complement cell[i];
        for j = 0 to n−1, j ! = i
        { read cell[i]; read cell[j]; }
        complement cell[i];
        }
```

3: write 1 in all cells;

4: replay Step 2;

}

The **Walking pattern algorithm** is similar to GALPAT except that the base cell is read only after all others are read. There are alternatives to the GALPAT algorithm, such as the **galloping diagonal, galloping row and galloping column**. In these algorithms, instead of shifting a 1 through the memory, a complete diagonal, row, or column of 1's is shifted. The **sliding diagonal/row/column** algorithm is similar to the galloping diagonal/row/column algorithm, but only those cells that are supposed to contain 1 are read after each shift. The **butterfly algorithm** is modified from GALPAT, with the purpose to find only Address Decoder Faults and Stuck-At Faults. In the **moving inversion (MOVI) algorithm**, the memory is initialized to all 0's, then this string of 0's is successively inverted to become all 1's, and vice versa. It ensures that no cell is disturbed by a read/write operation on another unrelated cell, and it detects all Address Decoder Faults and Stuck-At Faults. The algorithm is described below:

Procedure Butterfly
{

1: write 0 in all cells;

2: for i = 0 to n−1

{ complement cell i;

dist = 1;

while dist <= maxdist

/* maxdist < 0.5* col/row length */

```
        {
        read cell at dist north from cell[i];
        read cell at dist east from cell[i];
        read cell at dist south from cell[i];
        read cell at dist west from cell[i];
        read cell[i];
        dist *= 2; /* or dist += skip */
        }
        complement cell[i]; }
        3: write 1 in all cells;
        4: replay Step 2;
}
```

Surround disturb algorithms attempt to examine how the cells in a particular row/column are affected when complementary data is written into adjacent cells of nearby rows/columns.

Various bit oriented March test algorithms such as MATS, MATS+, Marching 1/0, MATS++, March X, March C, March C-, March A, March X and March B are summarized in [14]. Each of these algorithms performs a series of actions so that various defects in the memory arrays can be tested. For example, **March C-** is given by the following March elements:

↕(w0); ⇑(r0,w1); ⇑(r1,w0); ⇓(r0,w1); ⇓(r1,w0); ↕(r0)

These elements form a series of operation to be done as follows:

a) Write 0 in all memory cells in any address order

b) Read expected 0 from memory cells in increasing order of their address. And write 1 at the same time

c) Read expected 1 from memory cells in increasing order of their address. And write 0 at the same time

d) Read expected 0 from memory cells in decreasing order of their address, i.e. form the highest address to the lowest address. And write 1 at the same time

e) Read expected 1 from memory cells in decreasing order of their address. And write 0 at the same time

f) Read expected 0 from all the cells in any order.

March C− is known to completely detect Stuck at faults, Address decoder faults, AFs, unlinked transition faults, and coupling faults.

Other March patterns [4, 14] are shown below:

Modified algorithmic test sequence (MATS): { ↕(w0); ↕(r0,w1); ↕ (r1)}

MATS+: {↕(w0); ⇑(r0,w1); ⇓(r1,w0)}

Marching 1/0: {⇑(w0); ⇑(r0,w1,r1); ⇓(r1,w0,r0); ⇑(w1); ⇑(r1,w0,r0); ⇓(r0,w1, r1)}

MATS++: { ↕(w0); ⇑(r0,w1); ⇓(r1,w0,r0)}

March X: { ↕(w0); ⇑(r0,w1); ⇓(r1,w0); ↕(r0)}

March C: { ↕(w0); ⇑(r0,w1); ⇑(r1,w0); ↕(r0); ⇓(r0,w1); ⇓(r1,w0); ↕(r0)}

March A: {↕(w0); ⇑(r0,w1,w0,w1); ⇑(r1,w0,w1); ⇓(r1,w0,w1,w0); ⇓(r0, w1,w0)}

March Y: {↕ (w0); ⇑(r0,w1,r1); ⇓(r1,w0,r0); ↕(r0)}

March B: {↕(w0); ⇑ (r0,w1,r1,w0,r0,w1); ⇑ (r1,w0,w1); ⇓ (r1,w0,w1,w0); ⇓ (r0,w1,w0)}

A March test for word-oriented memories is constructed starting from tests for bit oriented memories [7]. These March patterns are shown to test static and dynamic faults in RAM [15].

Macrotest [16] was introduced to transfer handcrafted functional test patterns to embedded memory arrays via embedding in scan tests. This is useful when MBIST is too expensive. However, this technique is not a delay test. Scan test is used to test latch-based embedded arrays [17] for stuck-at, stuck-open and bridging faults. However, no delay tests are used in this approach to test the memory. Higher fault coverage of non-scan cells using at-speed functional patterns has been achieved [18]. The logic surrounding the memory can be tested using compressed scan-based testing [19] but the memory is still tested using BIST/functional patterns.

A seven-valued algebra is used to resolve the non-scan elements [20]. However, it is seen that the fault coverage and the test time rises with use of this technique.

1.5 Structure of thesis

Pseudo Functional Test for K Longest Path per Gate (PKLPG) is introduced in Chapter II. In this chapter, we discuss the ATPG tool *CodGen* which generates patterns

to detect the K longest paths through each line in a circuit. Preamble cycles and path generation are discussed in detail.

Our proposal to test the memory is described in Chapter III. In this chapter, we discuss how the memory arrays are tested using the present infrastructure and what needs to be added. Implementation details are described in Chapter IV. Chapter IV also summarizes the types of memory arrays that can be tested and impact of testing larger memory arrays. The results are discussed in Chapter V followed by conclusions and future work in Chapter VI.

2. PSEUDO FUNCTIONAL TEST FOR K LONGEST PATH PER GATE

2.1 Pseudo functional test (PFT)

Pseudo Functional Test (PFT) is a class of structural delay tests, in which traditional launch-on-capture delay testing is extended to additional launch and capture cycles. A test pattern is scanned into the circuit, and then multiple functional clock cycles are applied to it, with at-speed launch and capture for the last two cycles. The circuit switching activity over an extended period allows the off-chip power supply noise transient to die down prior to the at-speed launch and capture. This helps us in reaching an operating environment close to functional mode when the launch and capture take place. This increases the delay test correlation between structural and functional models, and minimizes over and under testing [21].

However, structural delay tests like scan based test and PFT use a slow-fast-slow clocking approach, with slow scan and fast functional clock cycles. In launch-on-capture (LOC) test, the time delay between the last scan-in cycle and the first functional cycle is long enough to allow the scan enable (SE) signal to change. The scan cycle time is typically an order of magnitude slower than the functional cycle time, in order to minimize the area and power cost of scan chain routing and buffering. Since the power grid is designed for functional operation, it largely reaches its quiescent state prior to the first functional (launch) cycle. The power grid time constant due to off-chip inductance is much longer than the functional clock cycle, so it takes dozens of functional clocks before the inductor currents can ramp up to supply the on-chip switching activity. In the meantime, this current must be supplied from on-chip parasitic and decoupling

capacitance, causing the supply voltage to droop. This is known as the *dI/dt* effect, and is the dominant power supply noise problem during delay test. This initial voltage droop causes the circuit to operate more slowly than in normal functional operation [22, 23]. Essentially, there is a mismatch between scan and functional test speeds due to the mismatch between the functional and test power supply voltages.

However, PFT can also be used to address the *dI/dt* effect. One solution to the *dI/dt* problem is to apply a series of scan or functional cycles that are slower than functional speed, but much faster than scan speed, to ramp inductor currents prior to the launch and capture of the delay test [22, 23]. These cycles are termed *preamble cycles*, as shown in Figure 2.

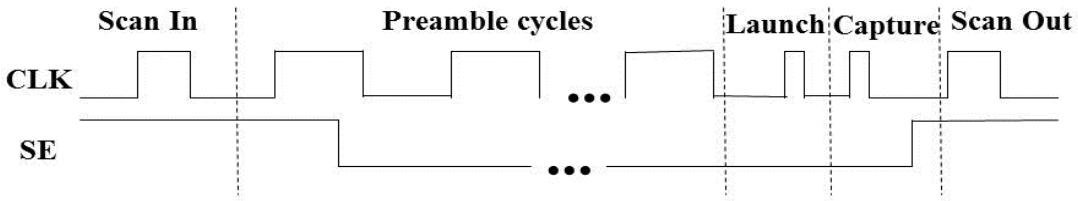

Figure 2 Preamble cycles followed by at-speed launch and capture cycles in PFT

The preamble must have following characteristics:

1. Significantly longer than the chip-to-package power grid time constant, so that transients have died down.

2. Clock cycle time significantly shorter than the off-chip time constant, so that the off-chip supply can reach steady-state. Figure 2 shows a constant preamble cycle

time, but it can be ramped down towards the functional cycle time, to minimize the voltage transient.

3. Propagate launch transitions on the shortest paths to minimize the chances of test invalidation due to a delay fault caused by supply noise during preamble cycles.

4. Weighted switching activity (WSA) per unit time similar to full speed functional cycles, so the off-chip supply current can reach steady state.

5. As few preamble cycles as possible in order to minimize ATPG effort.

Preamble cycles can be viewed as time frame expansion of normal 2 frames of LOC as shown in Figure 3. So, the vector at each time frame is derived from the previous time frame until we get to time frame 0 where the vector is scanned in through the scan chain. If any of the vectors cannot be derived from the previous time frame, then such vectors are discarded.

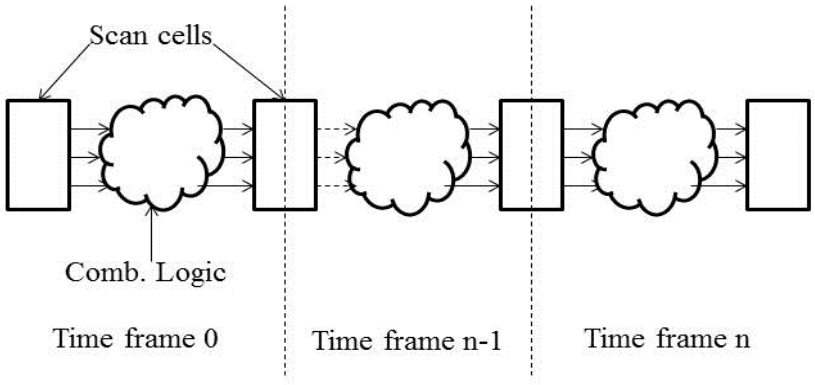

Figure 3 Time frame expansion of the circuit for PFT

Similarly, a transition is launched using the last 2 time frames and the necessary assignments obtained in those two frames are justified to time frame 0 to obtain a pattern as shown in Figure 4.

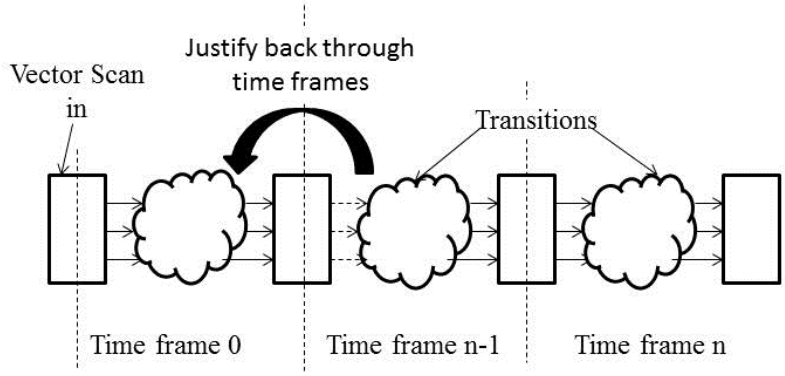

Figure 4 Extending the traditional LOC model for preamble cycles in PFT

2.2 K longest path per gate (KLPG)

The KLPG algorithm generates the K longest rising and falling paths through a target line under robustness constraints. The search space for each fault site, as shown in Figure 5 shows the fan-in and fan-out paths of the target line.

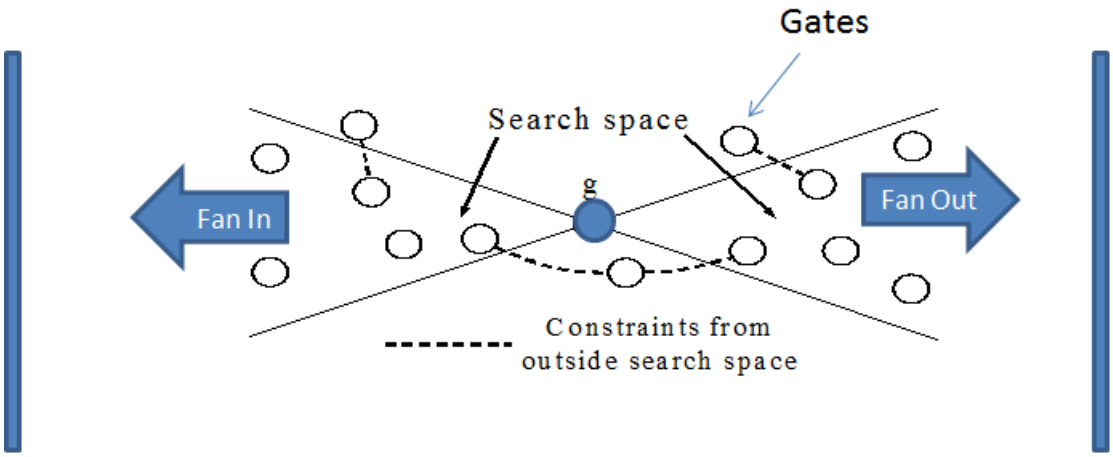

Figure 5 KLPG path search space

There are three major steps in the KLPG process:

1. Path Initialization: The setup of the Launch Points in the circuit. A launch point can be a Primary Input (PI) or a Pseudo Primary Input (PPI). A PPI is a scan cell output.

2. Path Growth: Extending the path by adding one gate at a time, so that it extends from a Launch Point to a Capture Point. A Capture Point can be a Primary Output (PO) or Pseudo Primary Output (PPO). A PPO is a scan cell input.

3. Final Justification: Finding the test pattern as well as checking compatibility between all internal assigned values in the circuit.

The overall summary of the KLPG algorithm is shown in Figure 6.

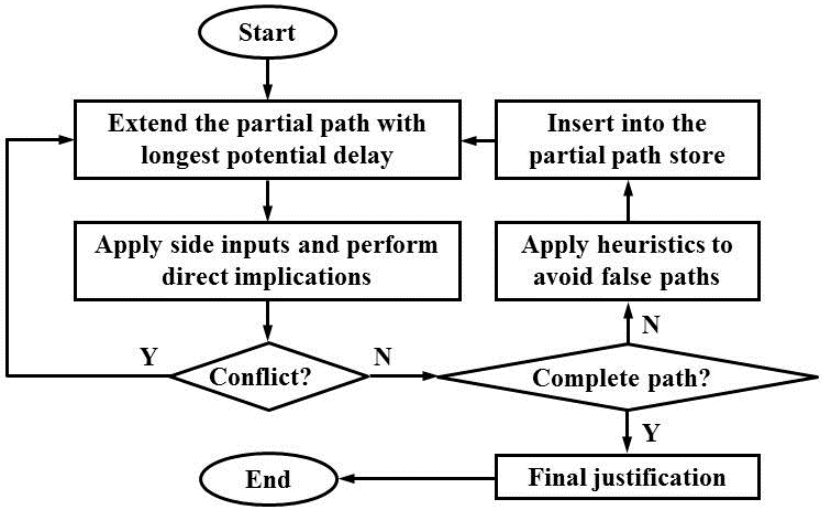

Figure 6 Overview of KLPG Path Generation

The KLPG algorithm is given below:

1. Parse the input files and perform pre-processing steps such as computing SCOAP metrics on each gate.

2. **For** each target fault site, until K paths has been generated or no more are possible

3. Initialize the paths from the target fan-in cone Launch Points.

4. Add these to the Partial Path Store.

5. Extract the partial path with maximum *Esperance*.

6. Extend the extracted path.

7. Add side input constraints and perform Multi-Frame Direct Implications.

8. **If** a conflict is detected

9. Delete this partial path

10. **End If**

11. **Else If** Complete path formed

12. Perform Final Justification.

13. **End If**

14. **Else If** Complete Path is not formed

15. Apply false path elimination heuristics.

16. Update the *Esperance*.

17. Re-insert in a sorted fashion into the Partial Path Store

18. Go to step 6.

19. **End If**

20. **End For**

3. TESTING THROUGH MEMORY ARRAYS

3.1 Memory as a logical model

CodGen can generate patterns for K longest paths for each line in the given circuit. So if the memory is modeled as flip-flops and combinational logic, CodGen can handle such a circuit. Thus the logical model of memory can be tested using the standard CodGen infrastructure. However, as noted above, memory cells are non-scan cells. So CodGen must be modified to handle non-scan storage elements.

3.2 Non-scan cell in CodGen

During path generation, if any transition is captured in a non-scan cell, then it cannot be scanned out of the non-scan cell. The captured value needs to be propagated to an observation point, i.e. a scan cell. So an extra clock cycle is needed to propagate the Boolean value of any captured transition, as shown in Figure 7. The extra cycle is termed a *coda* cycle. The coda cycle is untimed (and so uses a slow clock) since we are just propagating the Boolean value.

However, the propagated value can be captured by another non-scan cell during the propagation cycle. In that case, the value is still not observable. So, the Boolean value needs to be propagated further so that it is captured in a scan cell. Thus, another coda cycle is necessary. Multiple coda cycles may be required if the propagated value keeps getting captured in a non-scan cell as shown in Figure 8. The value is propagated

until the path finds a scan cell. It may be infeasible to have a large number of coda cycles. So for a given test, the number of coda cycles is fixed and if a given path cannot be propagated to a scan cell even after propagating across this fixed number of coda cycles, then that path is dropped and deemed as a non-propagating path. A different test may try a different number of coda cycles.

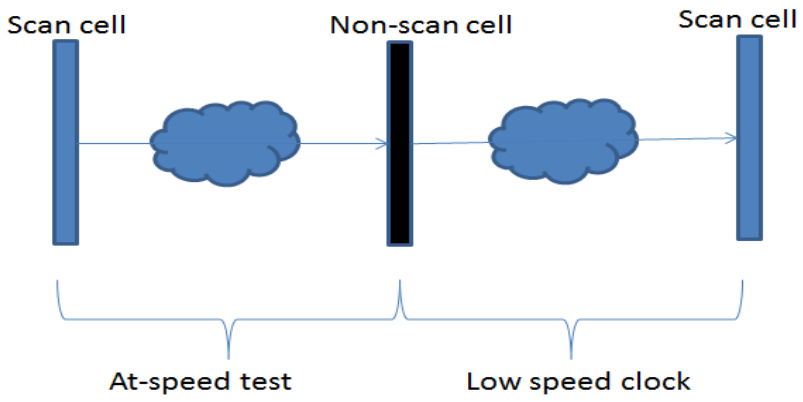

Figure 7 Propagation of Boolean value in coda cycle

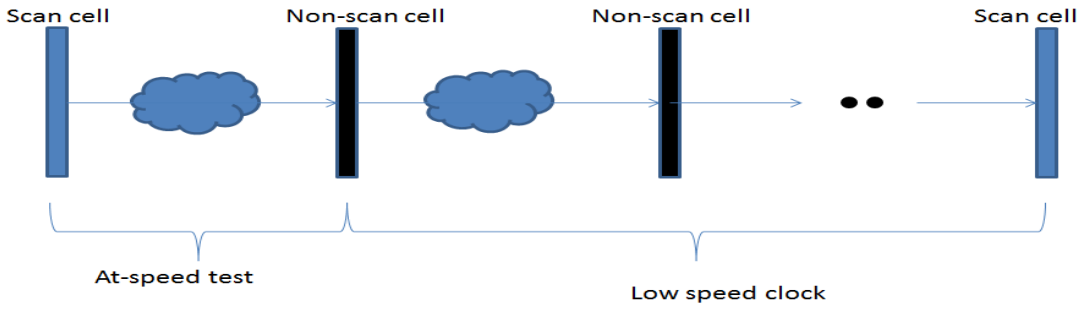

Figure 8 Multiple coda cycles

3.3 Coda cycle in CodGen

It was already discussed that the memory cells are non-scan cells and the captured transition is not observable. So, one extra *coda* cycle is introduced so that the Boolean value in the captured transition can be propagated to a scan cell. Thus, a number of coda cycles are added to the CodGen infrastructure for the propagation of the captured transition.

Figure 9 shows the modification to the PFT KLPG so that the non-scan memories can be tested. There are two functional test or at-speed cycles, launch and capture. In other words, there is just one capture cycle. The number of preamble cycles can be set to more than four. Similarly, the number coda cycles can be increased to more than 4. Once the propagated value is captured in a scan cell, scan enable is asserted and slow scan clocks are provided so that the captured value is shifted out of the scan chain.

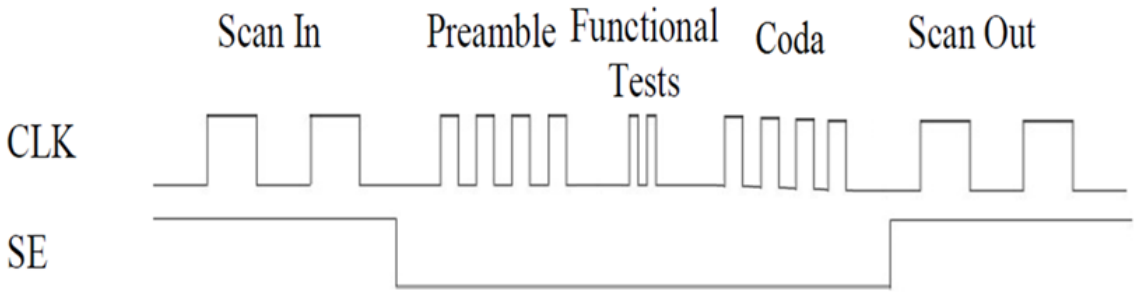

Figure 9 Coda cycle in PFT KLPG

The path to the non-scan cell is already tested in at-speed cycle once the value is captured in any non-scan cell. The propagation cycle, shown by the coda cycle, is just to read out the captured values and verify their correctness. So the coda cycles do not use the at-speed clock and can use a slower clock, but do not need to be as slow as the scan clock.

3.4 Easiest path propagation

The ATPG uses SCOAP metrics to first select the easiest (most observable, needing the fewest necessary assignments) path to a scan cell. If that fails, it tries the next-easiest, and so on. In this work, only a single coda cycle is implemented, which assumes there is only combinational logic between the memory (or non-scan cell) output and scan cells.

3.5 Writing into the memory

The longest path into a memory or non-scan cell is tested during the at-speed launch and capture cycles, as shown in Figure 7. These written values are propagated to the nearest scan cells in number of coda cycles.

3.6 Reading from the memory

Testing the longest paths out of the memory requires launching a transition at the memory output. One of the problems associated with black box memory models is that the memory outputs are X and so cannot produce transitions. Transitions cannot be launched when using memory bypassing because the outputs of the memory arrays are assigned to a fixed value. However, we are able to launch the transition because of the existing preamble cycles and use of memory arrays without bypassing or black boxing.

Suppose we have a 2x1 memory array with two cells, C0 and C1. We have one output data bit coming out of the memory array. The output wire is multiplexed between C0 and C1 so that the output data is selected based on the address fed to the select line of multiplexer, as seen in Figure 10.

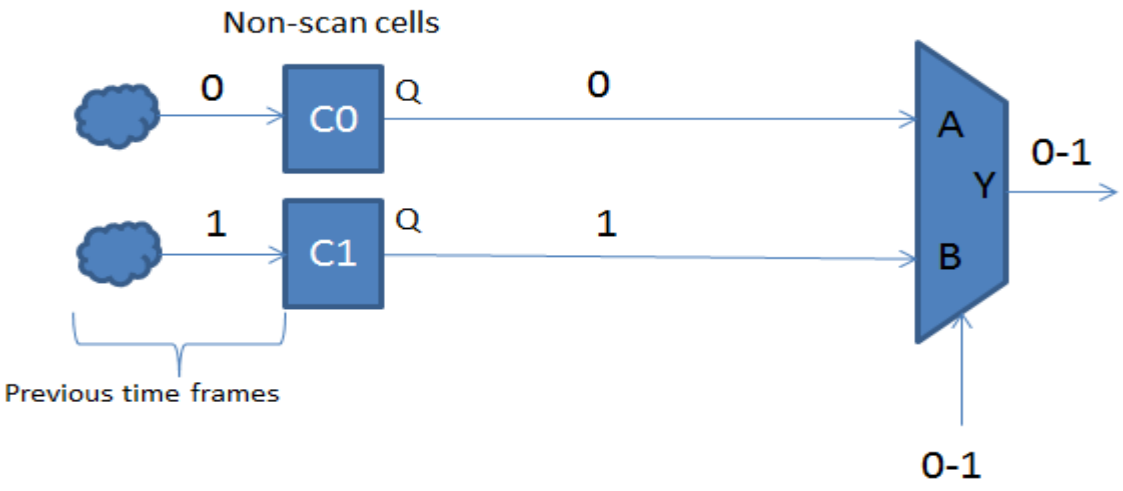

Figure 10 Launching a transition by toggling the address

If C0 holds 0 and C1 holds 1, if we want to launch a rising transition from the memory, then the address line switches from 0 to 1. Prior to the address transition, we must write C0 and C1 during the preamble cycles.

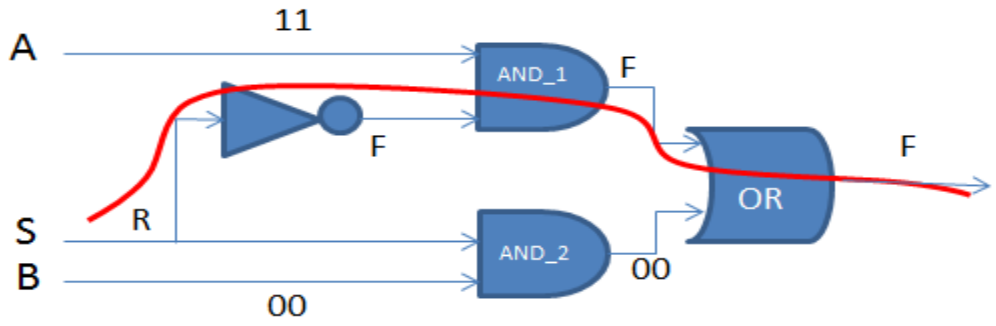

Figure 11 Necessary assignments inside a multiplexer to propagate a transition

The multiplexer shown in Figure 10 is further decomposed to primitive gates in Figure 11 to further describe the launch of a falling output transition F. The rising transition, given by R, starts from the select line, which is the address of the memory array. The path grows as it adds the NOT gate and the AND gate to its partial path. When the first AND gate is selected, the value of the non-controlling input of the AND gate has to be 11 in the last two time frames for robust test. Since the values 11 in input A are obtained from the non-scan cell C0, these values are written in preamble cycles so that they are available during at-speed cycles. The path further grows and adds OR gate in the partial path and it sets all the necessary assignments.

All the necessary assignments that are to be written to the non-scan cells are done in preamble cycles as shown in Figure 12. A sufficient number of preamble cycles are required to write all of the necessary values in non-scan cells.

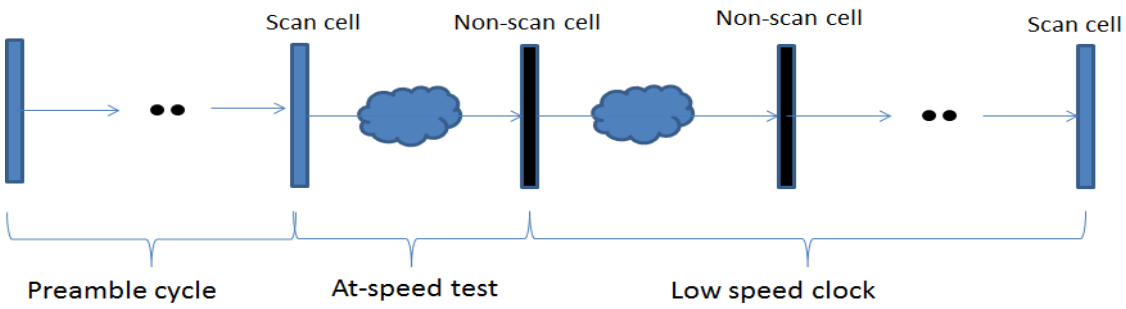

Figure 12 Preloaded values in preamble cycles

It is assumed that the memory is already preloaded with values before launching a transition from the memory. It is seen in Figure 11 that 0 and 1 have to be written in the two non-scan cells before a transition is launched during at-speed cycles. Thus, these values are written in preamble cycles which require at least two write cycles, since only one bit at a time can be written. In both write cycles, the write enable and chip enable must be asserted. In the first cycle, the address is set to the first non-scan cell and the data bit is set to 0. Then in the second cycle, the address is changed to the second non-scan cell and the data bit is set to 1. Thus it is required that there are at least two preamble cycles in this example.

4. IMPLEMENTATION

4.1 Memory model

There are various available memory implementations in industry and academia. Memory arrays can be combinational, high impedance when not read, etc. We have modeled the memory array with asynchronous read and synchronous write (using a global clock). In this model, data is written to the memory arrays during the edge of the clock when the correct address is decoded and write and chip enable signal are active low. This is similar to the implementation of a combinational RAM.

Typically when a value is written in a memory cell it might not be read for a long time because such memory cells are enabled by the address and the same address may not be decoded frequently. However, industry feedback suggests that the value written in cells will usually be read within the next few clock cycles. So we immediately read from the memory after writing the value into the memory. Furthermore, the memory is modeled as a uniform delay model such that all the delays are lumped at interface gates, with the same delay to and from every memory cell.

Large memories, such as L2 and L3 cache arrays, are not a uniform array, but a hierarchy of memory arrays and interconnection network. We assume that the network is already described and we just synthesize the memory blocks.

4.2 Synthesis

The behavioral description of the memory array is first defined in behavioral Verilog. The Verilog for 4x3 memory array is shown below:

```
`define addressBusSize  2
`define dataBusSize  3
module memoryArray4x3 (
input [`addressBusSize-1:0] A,//Address
input CLK,
input [`dataBusSize-1:0] D, // Data In
input EZ, // Enable chip, active low
input WZ,//Write enable , active low
output [`dataBusSize-1:0] Q // Data out
);
parameter depth=1<<`addressBusSize; //total depth = 2^addressBusSize
reg [`dataBusSize-1:0]reg_file[depth-1:0]; //total storage
//read operation. When chip enable is low
assign Q=(EZ==0)?reg_file[A]:`dataBusSize'bx;
//write operation. When both write enable and chip enable are low
always @ (negedge CLK)
begin
```

```
        if (WZ==0 && EZ==0)
                reg_file[A]<=D;
    end
    endmodule
```

The behavioral model is converted to a structural model using Synopsys Design Compiler. This tool synthesizes the behavior description to obtain a gate level implementation of the memory arrays. The logic model is shown in Figure 14. The general structure of the memory arrays can be thought as a memory cell and a feedback multiplexer as shown in Figure 13. If the correct address is decoded and write enable and chip enable are active low, new data is written to memory on the clock edge, else the same data loops around. The data can be read any time once the address is decoded.

Figure 13 General structure of a memory array

The model obtained may not look like the real memory, but this synthesis procedure can handle all memory types. The synthesized model is then flattened further

so that we get primitive gates from the complex logic gates like AOI (And Or Inverter), and XOR gates. This flattened model is then used to replace the memory black box in CodGen.

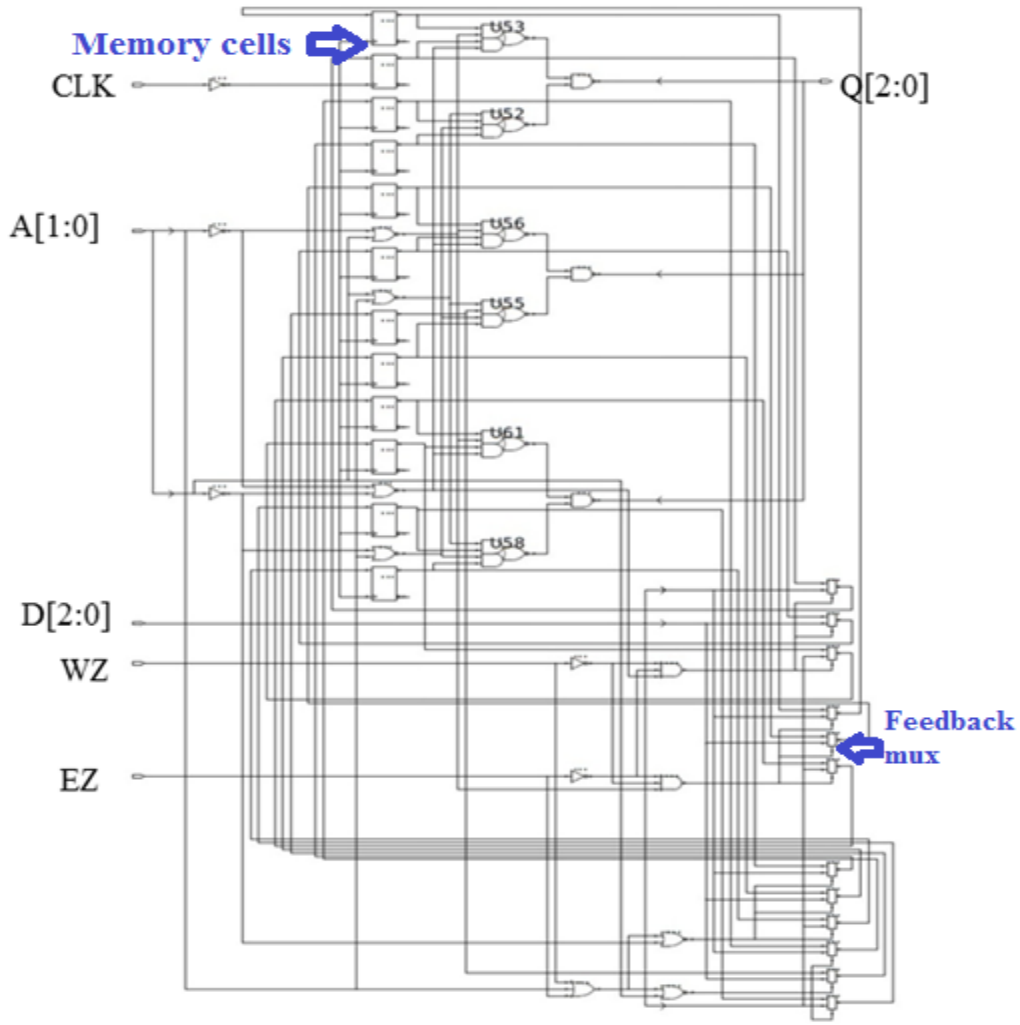

Figure 14 Logical model of 4x3 memory array

It can be seen in Figure 14 that there are a total of 12 non-scan cells, 4 address and 3 data bits. So, each address selects 4 non-scan cells. There are layers of gates for

writing the data into the memory and reading the data out of the memory. For reading the data from the memory, the first layer gives a total of 6 possible data bit values from 12 non-scan cells. The first layer is given by a series of AND-NOR gates given by U53, U52, U56, U55, U61 and U58. For example, U53 and U52 selects 1 output data from the first four scan cells. THe data to be selected depends on the decoded address that acts as an input to U53 or U52. Let us visualize the memory array as a 2D array of memory[4][3], where the first index is memory address and second index is the data bit. For any address, the asynchronous memory gives a 3-bit output. So U53 and U52 gives the value of memory[X][0] where X can be any address from 0 to 3 depending upon the decoded address. For any decoded address, U53 and U52 gives the 0^{th} bit of the data bit. It is noted that only one of U53 and U52 has a valid bit i.e. only one of them carry the required bit from the non-scan cell. U53 and U52 are then taken to an AND gate which is the second layer to read out the data. If U53 has valid data, then the output of U52 will be 1 and the data of U53 will be carried to the output. Similarly, while writing the data to the memory cell, there are layers of gates always followed by a feedback multiplexer as shown in Figure 13 and Figure 14.

As the number of address bits increases, the decoding logic becomes more complex and more layers of gates are required for decoding. Thus, with each additional address bit, an additional layer of decoding logic is added to the logic. One can easily think that with one additional bit of address, we need 1 more layer in Figure 14 so that we can select each data bit from the 8 addresses such that the data bits of the cells travel in a tree fashion as in a 4x3 memory array shown above. However, just adding an AND

gate with the addition of one extra address bit can be a naïve option. Thus, it is left to the tool, Synopsys Design Compiler, to choose the best combination of gates for decoding.

It is seen that the tool chooses various ranges of combinational gates to optimize the area and power of the circuit. Instead of using higher fan-in multiplexers, 2:1 or sometimes 3:1 multiplexers are used. Various combinational gates such as AND-OR-INVERT and OR-AND-INVERT are used. With the addition of more address and data bits, the gates are added in a tree to optimize the logic. Thus any size of memory can be synthesized.

Figure 15 and Figure 16 shows decoding of a 256x8 memory array. Figure 15 shows that the first data bit is decoded out of 256 possible addresses. The decoded address X is sent to the combinational logic and the 0^{th} data bit from the memory cell of the decoded address is obtained from the output of the 2:1 multiplexer. As mentioned above, the combinational logic depends upon the design library used by Synopsys Design Compiler and the type of optimization used, such as minimum power or minimum area. In our implementation, the default setting of medium power and minimum area is used. Figure 16 shows that there will be 8 instances where 8 data bits emerge. Each box in Figure 16 is equivalent to the whole circuit of Figure 15. The 8 bit output is the asynchronous output of the memory array.

The decoded address is obtained by similar combinational logic. There will be an increase in layers of combinational logic with an increase in address bits. These layers are also obtained in a tree nature similar to the tree of combinational logic while reading the data.

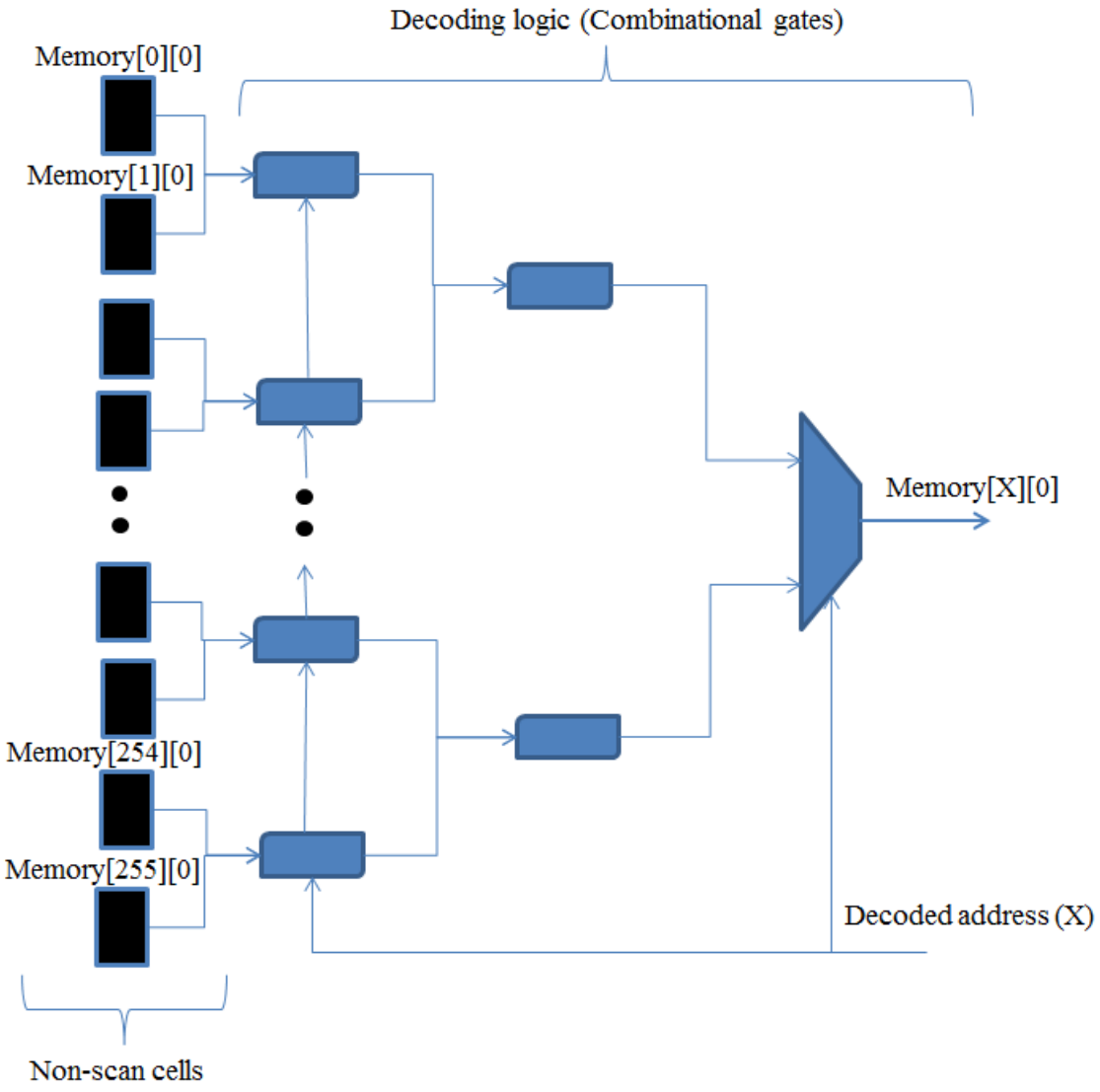

Figure 15 Decoding the LSB data bit of 256x8 memory

The typical RAM implementation is shown in Figure 17**.** It is seen that such an implementation also has a row decoder which is nothing but the address decoder which selects the correct row/address from the memory array. The data/column is then read out using a column decoder which is similar to the reading out of the data bits.

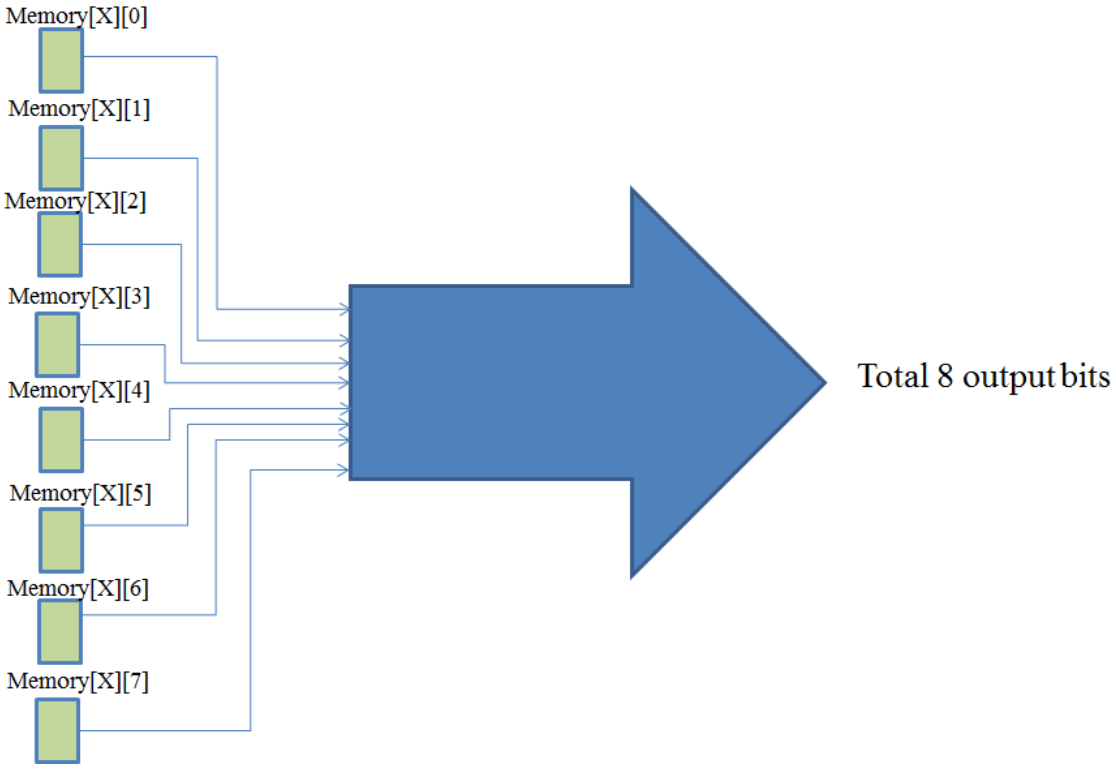

Figure 16 Concatenation of output data from the memory array 256x8

4.3 Resilience against process variations

The longest paths to and from the memory cells can vary due to process variations. There can be significant variation due to the minimum feature sizes used in dense memory arrays. Testing the K longest paths through each memory cell is robust against process variation.

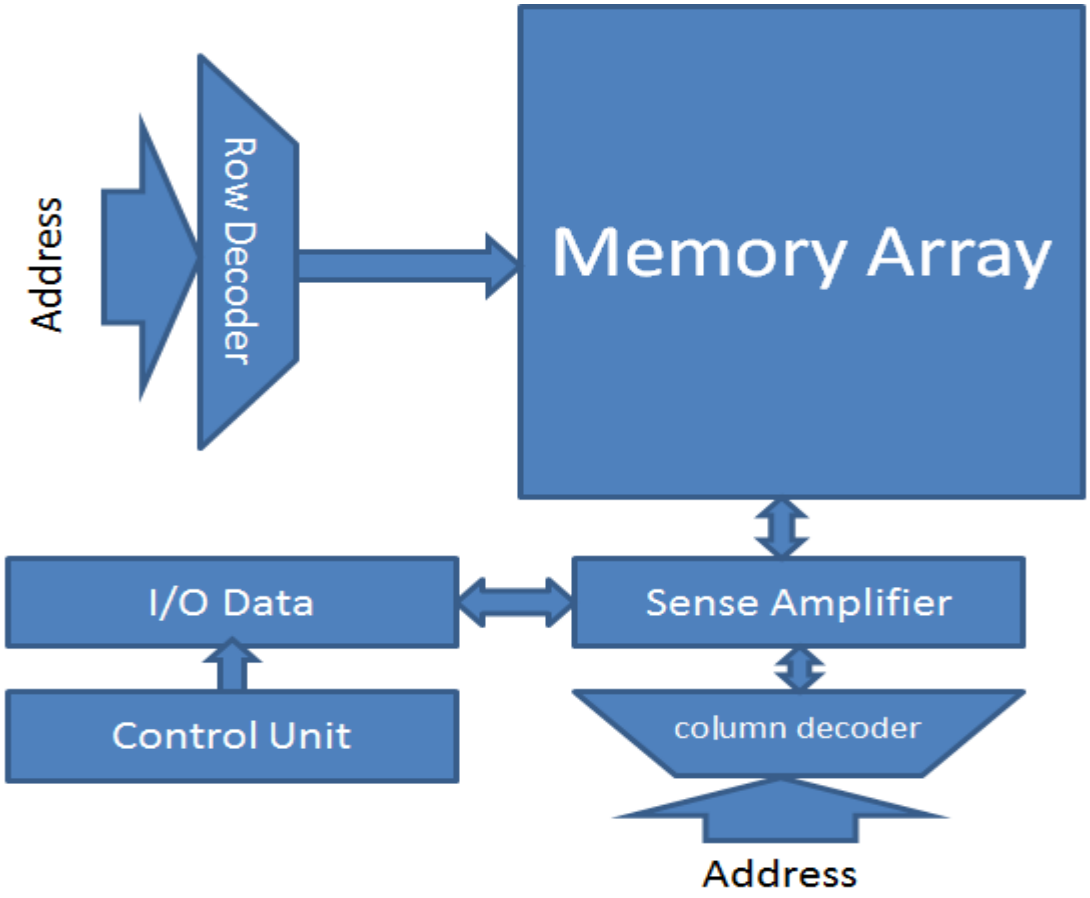

Figure 17 Typical implementation of SRAM

5. RESULTS

PFT KLPG for memory test is implemented in C++ running on a 1.86 GHz Intel Dual Core Processor with 8 GB of memory. Robust paths are used to generate patterns.

A total of 4 preamble cycles are used. Although 2 preambles are sufficient to preload values into non-scan cells, 4 cycles are used for better noise correlation. Only 1 coda cycle is used for the propagation of the Boolean value as it is assumed that the propagated value finds a scan cell within the next cycle. If any paths do not find a scan cell within that coda cycle, then such paths are discarded. So a scan chain is put on the memory output.

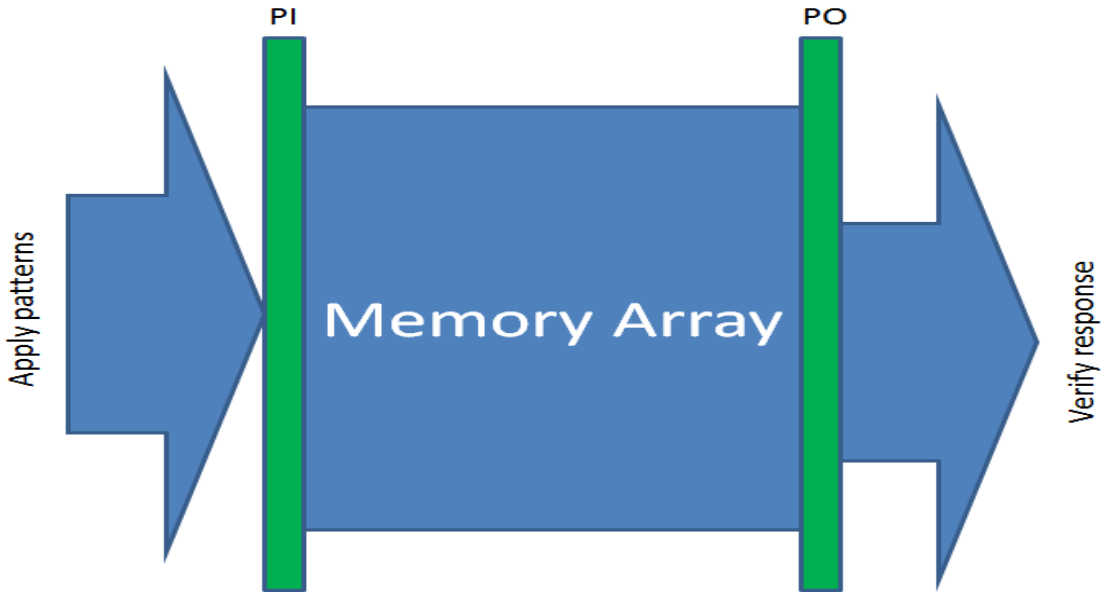

Figure 18 Standalone memory test

All the paths going into and out of the memory are tested. Thus, combined number of paths for writing into the memory and reading from the memory are generated. K equal to 1 is used, i.e. one longest and rising path per gate. Dynamic compaction is used so that all the patterns are compacted together. The memory arrays are tested as standalone memories i.e. PIs and POs of the memory arrays are directly tested as shown in Figure 18. However, the results obtained by using a scan wrapper around the memory arrays yield similar results.

There may be cases where a scan cell feeds its value to the non-scan cell through a combinational cloud and the propagated value passes through another combinational cloud to be captured in a scan cell as shown in Figure 19**.**

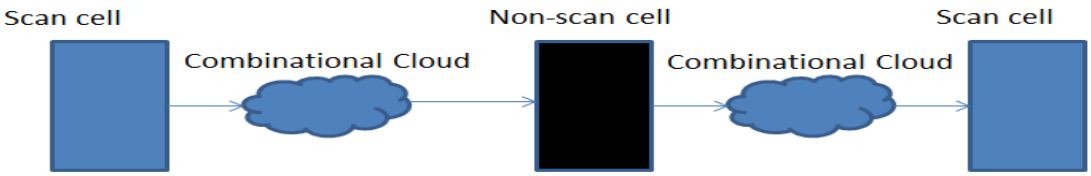

Figure 19 Scan cell around non-scan cell

The circuit can be reduced to Figure 20 which shows scan and non-scan cell without expanding the time frame. When the non-scan cells are expanded through the preamble cycles, they act functionally i.e. they are similar to scan cells except in the first and last time frames. In the first frame the non-scan cell has an unknown value and in

the last frame its output cannot be read out. Such scenarios are handled by the existing CodGen infrastructure.

Figure 20 Scan and non-scan cell before time frame expansion

Different memory arrays were synthesized and tested. The memory size is given as A X B where A denotes the total number of words and B denotes the number of output bits. Table 1 below shows results obtained from the PFT KLPG.

Table 1 Results of PFT KLPG standalone memory arrays

Memory Size	Paths	Patterns	Time	Gate count
2x1	15	6	0:00:00	20
4x3	58	16	0:00:02	86
4x8	232	16	0:00:05	275
8x4	227	33	0:00:07	255
16x8	1022	73	0:01:10	1167
8x16	1044	34	0:00:48	1160
64x8	4161	305	0:13:53	4686
8x64	4116	35	0:09:04	4568
128x8	8428	608	1:01:49	9419
8x128	8212	34	0:33:00	9112
256x8	16661	1114	3:30:17	18726
8x256	16404	33	1:40:24	18200

The patterns generated are sufficient to test all of the paths in the memory arrays. In the 8x256 memory, the paths can be tested by writing eight words of 0s and eight words of 1s and reading can be tested by reading those eight words of 0s and eight words of 1s. So, 32 patterns are necessary to test the memory. CodGen generates total of 33 patterns. A 256x8 memory can be tested by 1024 patterns of 8-bits each, while dynamic compaction achieves 1114 patterns. The first reason that CodGen does not find the minimum number of test patterns is that the paths are generated for each bit first rather than the whole word. For example, in the 8x256 array, the paths are not generated for the 256 bits at once, rather for each data bit of a word. Thus this greedy nature of the KLPG compaction algorithm led to the increase in number of patterns. The other reason is the path pool size. It is seen that there are over 16k paths for both 8x256 and 256x8 and the path pool size is 2000. So dynamic compaction has to write patterns to memory before some paths are considered.

The paths, pattern and gate count increase approximately linearly with the increase in memory size. The number of paths is slightly more than linear and number of gates slightly less than linear due to the select logic overhead. However, CPU time increases super linearly with circuit size. CPU time for the 8x16 memory array is 31 seconds and for the 8x64 memory array is 6 minutes 17 seconds. Thus the time increases by more than 6 times when the circuit size increases by 4 times. The total CPU time for path justification increases by 12 times, path generation by 13 times and dynamic compaction by 18 times. Since the 8x64 memory just adds four times as many paths that are independent (except for the enable signals), the CPU times for each of these phases

should have only increased by 4 times. Dynamic compaction CPU time is super linear in pattern count, but the compacted pattern count is nearly the same for both memories. Path generation consumes 92% of the total CPU time, so it is the dominant reason for super linear CPU time.

Total CPU time comprises of ATPG time, which takes majority of the time and Dynamic Compaction time. It is seen that in ATPG time, recording the path, direct implication and generation of paths take more than half of the ATPG time. For example, in 8x64 memory array, ATPG takes 335 CPU seconds and Dynamic Compaction takes 116 CPU seconds. ATPG CPU seconds is divided as recording path-132, direct implication-76, generation of path-62, FAN -23, propagation time-14 and rest of the time is taken for initial setup. For 8x16 memory array, ATPG takes 26 CPU seconds and Dynamic Compaction takes 6 CPU seconds. ATPG CPU sec is divided as recording path-8, direct implication-5, generation of path-5, FAN-1, propagation time - 1 and rest of the time is taken for initial setup. Thus superlinear nature of the CPU time is caused by various fragments of the code.

The patterns generated in the above table are for standalone memories i.e. those memory arrays which do not have any combinational logic surrounding them as shown in Figure 21. However, when there is a combinational cloud around the memory arrays, the number of paths should decrease. Writing and reading 0s and 1s from all the addresses may not be possible due to the logic constraints presented by the combinational cloud. Furthermore, the Boolean value might fail during propagation. We test a memory array with a combinational cloud before the input to the memory.

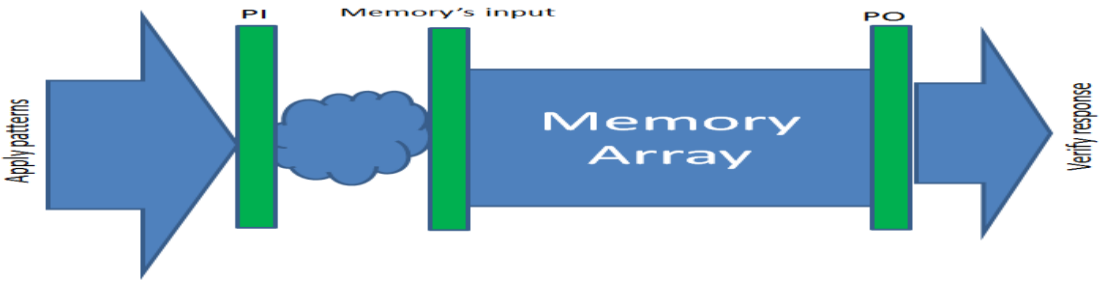

Figure 21 Combinational cloud before memory array

In our experiment, several of the PIs of the circuit are fed through a full adder before entering the memory array. For small circuits like 2x1, 4x3 and 4x8 which have less than 2 address lines, a half adder is used. For a 2x1 memory, the output of the half adder is connected the address line and the write enable line. For other models, the output of the adder is connected to the address lines only. The results obtained from PFT KLPG are shown in Table 2.

Table 2 Results of PFT KLPG of memory arrays with combinational cloud

Memory Size	Paths	Patterns	Time	Gate count
2x1	20	14	0:00:01	26
4x3	50	17	0:00:01	92
4x8	182	17	0:00:03	281
8x4	195	39	0:00:05	269
16x8	831	76	0:00:38	1181
8x16	810	43	0:00:36	1174
64x8	3355	320	0:10:09	4700
8x64	3163	43	0:07:05	4582
128x8	6781	595	0:46:56	9433
8x128	6298	44	0:27:12	9126
256x8	12528	996	3:38:29	18726
8x256	16404	33	1:40:24	18214

It is seen that the number of paths have decreased when combinational logic is introduced in front of the memory arrays. However, there is an increase in pattern count in some circuits. This is because of the fact that the logic may introduce more constraints that preclude compacting paths together into the same test pattern. However in the 256x8 memory, there is a decrease in pattern count the adder significantly reduces the number of testable paths (from 16661 to 12528).

6. CONCLUSIONS AND FUTURE WORK

PFT KLPG test generation is used to test the longest paths to and from all cells in a memory array. These patterns perform path delay tests so that critical paths through the memory can be tested. This approach avoids any delay or area overhead incurred by MBIST or loss in fault coverage incurred by treating memory as block box or just performing functional tests to the memory. The approach has been demonstrated with combinational logic constraints on the address input of the memory.

Future work includes using a template to model the memory arrays as shown in Figure 22. The template has a common bus for input and output data. The output from each memory cell passes through a tristate buffer which is enabled when the correct address is decoded and output enable is active. The data is written to the cells at a clock edge when write enable is active and correct address is decoded. Unlike the design used in this research, the clock is gated so that only the word being written is clocked. These templates have a closer resemblance to physical memory implementations and their switching activity can better correlate with physical memory and with the power model supplied with the memory.

We assume delays are lumped at interface and future work includes adding delays such that the logical model delays match that of the behavioral delay model supplied with the memory array.

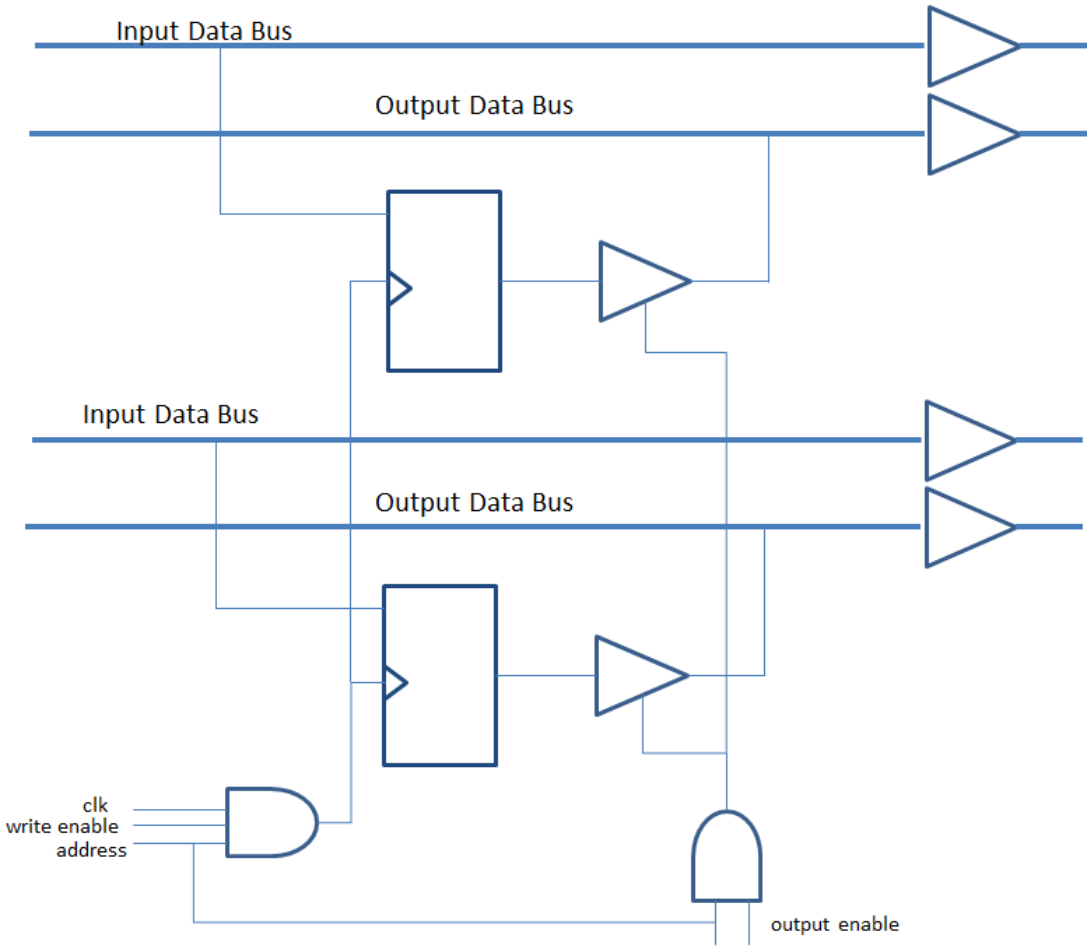

Figure 22 Memory template

REFERENCES

[1] Belete, D., Razdan, A., Schwarz, W., Raina, R., Hawkins, C., & Morehead, J. (2002). Use of DFT techniques in speed grading a 1 GHz+ microprocessor. In *Test Conference, 2002. Proceedings. International* (pp. 1111-1119). IEEE.

[2] Zeng, J., & Abadir, M. (2004, April). On correlating structural tests with functional tests for speed binning. In *Current and Defect Based Testing, 2004. DBT 2004. Proceedings. 2004 IEEE International Workshop on* (pp. 79-83). IEEE.

[3] Crouch, A. L. (1999). *Design-for-test for Digital IC's and Embedded Core Systems* (Vol. 1). Prentice Hall PTR.

[4] Wang, L. T., Wu, C. W., & Wen, X. (2006). *VLSI test principles and architectures: design for testability*. Academic Press.

[5] Van De Goor, A. J. (1993). Using march tests to test SRAMs. *Design & Test of Computers, IEEE*, *10*(1), 8-14.

[6] Dekker, R., Beenker, F., & Thijssen, L. (1988, September). Fault modeling and test algorithm development for static random access memories. In *Test Conference, 1988. Proceedings. New Frontiers in Testing, International* (pp. 343-352). IEEE.

[7] Van de Goor, A. J., & Tlili, I. B. (1998, February). March tests for word-oriented memories. In *Design, Automation and Test in Europe, 1998., Proceedings* (pp. 501-508). IEEE.

[8] Abadir, M. S., & Reghbati, H. K. (1983). Functional testing of semiconductor random access memories. *ACM Computing Surveys (CSUR)*, *15*(3), 175-198.

[9] Zarrineh, K., Upadhyaya, S. J., & Chakravarty, S. (1998, October). A new framework for generating optimal march tests for memory arrays. In *Test Conference, 1998. Proceedings., International* (pp. 73-82). IEEE.

[10] Rajsuman, R. (2001). Design and test of large embedded memories: An overview. *IEEE Design & Test*, *18*(3), 16-27.

[11] Zarrineh, K., & Upadhyaya, S. J. (1999, January). On programmable memory built-in self test architectures. In *Proceedings of the conference on Design, automation and test in Europe* (p. 136). ACM.

[12] Crouch, A. L., McKeown, J. L., & Shepard, C. G. (1999). *U.S. Patent No. 5,995,731*. Washington, DC: U.S. Patent and Trademark Office.

[13] Tuna, M., Benabdenbi, M., & Greiner, A. (2007, May). At-Speed Testing of Core-Based System-on-Chip Using an Embedded Micro-Tester. In *VLSI Test Symposium, 2007. 25th IEEE* (pp. 447-454). IEEE.

[14] Van de Goor, A. J. (1991). *Testing semiconductor memories: theory and practice*. John Wiley & Sons, Inc..

[15] Hamdioui, S., Al-Ars, Z., & van de Goor, A. J. (2002). Testing static and dynamic faults in random access memories. In *VLSI Test Symposium, 2002.(VTS 2002). Proceedings 20th IEEE* (pp. 395-400). IEEE.

[16] Ross, D. E., Wood, T., & Giles, G. (2000). Conversion of small functional test sets of nonscan blocks to scan patterns. In *Test Conference, 2000. Proceedings. International* (pp. 691-700). IEEE.

[17] Yang, F., & Chakravarty, S. (2010, November). Testing of latch based embedded arrays using scan tests. In *Test Conference (ITC), 2010 IEEE International* (pp. 1-10). IEEE.

[18] Banga, M., Rahagude, N., & Hsiao, M. S. (2011, March). Design-for-test methodology for non-scan at-speed testing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011* (pp. 1-6). IEEE.

[19] Seok, G., Kim, H., & Mohammad, B. (2012, April). Write-through method for embedded memory with compression Scan-based testing. In *VLSI Test Symposium (VTS), 2012 IEEE 30th* (pp. 158-163). IEEE.

[20] Qiu, W., Wang, J., Walker, D. M. H., Reddy, D., Lu, X., Li, Z., & Balachandran, H. (2004, October). K longest paths per gate (KLPG) test generation for scan-based sequential circuits. In *Test Conference, 2004. Proceedings. ITC 2004. International* (pp. 223-231). IEEE.

[21] Lahiri, S. (2013). Pseudofunctional Delay Tests For High Quality Small Delay Defect Testing.

[22] Nadeau-Dostie, B., Takeshita, K., & Cote, J. F. (2008, October). Power-aware at-speed scan test methodology for circuits with synchronous clocks. In *Test Conference, 2008. ITC 2008. IEEE International* (pp. 1-10). IEEE.

[23] Pant, P., & Zelman, J. (2009, May). Understanding Power Supply Droop during At-Speed Scan Testing. In *VLSI Test Symposium, 2009. VTS'09. 27th IEEE*(pp. 227-232). IEEE.